Garrett Hall
CSC 173
Scheme Weeks 3-4

1.0 Overview
I organized this document into 6 sections which do not necessarily have to be read in order. First I'll describe the state and node representation, then how problems are generated and solved. The last sections are devoted to the visualization of search trees and statistical results.

2.0 States and Nodes
The data structures, successor functions, and nuts-and-bolts of filtering illegal and backwards states are discussed here.

2.1 Nodes
Nodes are required to represent a search tree. Each node is defined as a structure containing four values—pointer to the parent, action, depth, and state.

```
(define-struct node (parent action depth state))
```

The action value is extraneous, as actions can be inferred if we traverse up the search tree and observe differences between states. However, it is useful for debugging purposes.

2.2 State Representation
Nodes only serve as a data structure for the explored state space. The representation of states and the transitions between them are what actually determine the tree structure. States are defined by vectors of dimension n^2—the 0 component representing the blank square. For $n = 3$ the code to create a *root node* would be

```
(define State #(6 0 8 1 2 3 4 5 7))
(define MyRoot (make-node null 'Root 0 State))
```

The `make-node` constructer is provided by Scheme. Instantiating the last "action" to `Root` and depth to `0`, implies `MyRoot` is a *root node*. To output anything in readable form, use the `print` function

```
> (print MyRoot)
0 Root
{{6,0,8},
{1,2,3},
{4,5,7}}
```

Or graphically,

```
> (mm-print MyRoot)
```

0 Root

6		8
1	2	3
4	5	7

Note: All graphics were created in Mathematica. In addition to normal textual output, I've written printing functions for Mathematica. To see what output looks like as Mathematica code, simply add the prefix "`mm-`" *to print functions. In the future when graphics appear, they have been automatically generated from Scheme output and not "hand-drawn".*

2.3 State Transitions

Since there are four possible actions—up, down, left, right—there will be four possible successor states to any given state. These can be found through simple *transformation* functions where *b* is the index of the blank and n^2 is the size of the puzzle:

$$Up = \text{VectorSwap}(b,\ b-n)$$
$$Down = \text{VectorSwap}(b,\ b+n)$$
$$Left = \text{VectorSwap}(b,\ b-1)$$
$$Right = \text{VectorSwap}(b,\ b+1)$$

However, we don't want to generate illegal states. Filtering backwards-moving states, depth-limited states, and closed-list states may also be desirable. To see how this works, look at the successor function:

```
(define (valid-successors n closed max-depth)
    (map (lambda (a t f1 f2 f3)
           (if (or f1 f2 f3) null (new-node n closed a t)))
         '(U D L R)
         (transform (blank (node-state n)))
         (filter-valid (blank (node-state n)))
         (filter-backwards (node-action n))
         (filter-deep (node-depth n) max-depth)))
```

The logic is simple. Map each move `(U D L R)`, its transformation `t`, and its filters `f1 f2 f3` to a lambda function. If none of the three filters apply, generate the new node. Otherwise the node is not generated at all.

For example given the previously discussed state of `MyRoot`:

0 Root

6		8
1	2	3
4	5	7

- `filter-valid` will generate `(#t #f #f #f)`, meaning filter out the `U` (up) move because it is impossible unless the blank square leaves the border of the puzzle.
- `filter-backwards` will filter out any "backwards" moves that revert to the previous state. (In this case there are no previous states so it does not apply.)
- `filter-deep` will filter out all nodes, generating either `(#t #t #t #t)` or `(#f #f #f #f)` if the depth has gone beyond the `max-depth` argument.

To generate the successors to the node `MyRoot` and print a list of them:

```
(define Depth-Limit 2)
(define Succ (successors MyRoot (new-closed-list) Depth-Limit))
> (print Succ)
```

1 D

6	2	8
1		3
4	5	7

1 L

	6	8
1	2	3
4	5	7

1 R

6	8	
1	2	3
4	5	7

The `U` move has been filtered. To grab the successor node in the middle (`1L` obtained by moving left) and print it:

```
(define 1L (list-ref Succ 1))
> (print 1L)
```

1 L

	6	8
1	2	3
4	5	7

0 Root

6		8
1	2	3
4	5	7

The traversal of the tree from `1L` to `MyRoot` gets printed. Printing the successors of `1L` gives:

```
(define FILTER-BACKWARDS #t)
> (print (successors 1L (new-closed-list) Depth-Limit))
```

2 D

1	6	8
	2	3
4	5	7

Note that the backwards move to the right has been filtered by setting global flag `FILTER-BACKWARDS` to true. If instead, it is set to false

```
(define FILTER-BACKWARDS #f)
> (print (successors 1L (new-closed-list) Depth-Limit))
```

2 D

1	6	8
	2	3
4	5	7

2 R

6		8
1	2	3
4	5	7

Both successor nodes are generated as one might expect.

3.0 Generating Problems and Solutions

Efficiently generating problems of varying difficultly is not as trivial as first appears. Once a problem is created, I'll demonstrate how to run a search on it and interpret the results.

3.1 Generating Problems

With the details of the successor function out of the way, it is not too difficult to begin generating problems and using naïve search functions to solve them. A problem is a random root node. To generate small, illustrative problems I've chosen $n = 3$. However, the global variable `N` can be changed independently of any other part of the code (so long as the definitions are reloaded).

```
(define N 2)
(load "nodes")
(print (random-root 10))

(define N 5)
(load "nodes")
(print (random-root 1000))
```

0 Root

3	1
2	

0 Root

8	7	4	3	6
2	1	21	24	9
	11	20	15	10
22	23	13	18	14
19	12	17	16	5

Starting from the goal state, the function `(random-root n)` simply does a random walk of `n` moves. To get a random root so that the minimal cost solution is *exactly* `n` moves, `random-root-checked` will generate random nodes and check the solution path using IDS.

This method of generating problems is preferable to "true" random problems obtainable through random permutation for two reasons. First, random permutations need to be parity-checked to ensure they are solvable (half of all random permutations have no solution). Second, manipulating solution *depth* rather than the size of *n* provides better control over the difficultly of a problem.

Of course, a larger *n* will increase the probability distribution of solution depths. For instance it can be experimentally verified that `(random-root-checked 7)` does not terminate if $n = 2$ because it is impossible to generate a 3-puzzle that cannot be solved in less than 7 moves.

A larger *n* also increases the ratio of the number of nodes with four successors to nodes with three.

Breakdown of types of squares for puzzles of size *n*:

$CornerSquares = 4$

$EdgeSquares = 4n - 8$

$CenterSquares = n^2 - 4n + 4$

I will demonstrate changing *N* does not affect search times independently of depth in section 5.6. For the remainder of the examples we'll keep $n = 3$ unless otherwise stated.

3.2 Generating Solutions

As previously stated,

```
(define Root (random-root-checked 5))
```

will generate a problem with the optimal solution at *exactly* depth 5. To print the results of three different types of searches use:

```
(print (bfs Root 5))
(print (dfs Root 5))
(print (ids Root 5))
```

Each of the search functions takes the random node `Root` and a depth-limit as arguments. The output will be the solution path and some statistics about the search.

```
5 D
{{1,2,3},
{4,5,6},
{7,8,0}}

4 D
{{1,2,3},
{4,5,0},
{7,8,6}}

3 R
{{1,2,0},
{4,5,3},
{7,8,6}}

2 R
{{1,0,2},
{4,5,3},
{7,8,6}}

1 U
{{0,1,2},
{4,5,3},
{7,8,6}}

0 Root
{{4,1,2},
{0,5,3},
{7,8,6}}
```

BFS

```
Nodes expanded: 51
Tree size (nodes): 25
Closed-list size (states): 61
Search time (ms): 1
```

DFS

```
Nodes expanded: 42
Tree size (nodes): 8
Closed-list size (states): 44
Search time (ms): 0
```

IDS

```
Nodes expanded: 112
Tree size (nodes): 8
Closed-list size (states): 44
Search time (ms): 1
```

As the solution path results are identical for all three searches, I've only included them once. The statistics are for BFS, DFS, and IDS, respectively.

Although there are better methods for visualizing searches, a simple traversal output allows each to be checked for correctness. Notice that DFS outperforms IDS in the number of nodes expanded because IDS must run at depth-limits 1, 2, 3, and 4 before finding a solution.

The statistics are self-explanatory. Tree size is the final number of nodes in memory, including all nodes on the open list and their parents (counted once if shared). The number of nodes expanded is exactly the number of times the successor function has been called and therefore should be well-correlated with search time (all other things being equal.)

The search algorithms themselves are not particularly interesting. A single flag passed to a generic search function determines whether a BFS or DFS will be run by changing the behavior of the open list from a stack to a queue. The IDS just runs the DFS at increasing depth-limits. Most of the code is dedicated to generating statistics and visual output, which will be described next.

4.0 Visualizing Searches

In this section, I'll compare the tree structures of the different searches and how they are affected by depth-limits. Closed-list filtering will be described visually.

4.1 Comparing Search Trees

Although statistics quantify search results, the best way to visualize searches themselves is through tree connectivity. Sticking with an optimal goal at depth 5, but changing the maximum *accepted* goal depth— global variable `GOAL-DEPTH`— to 20:

```
(define GOAL-DEPTH 20)
(define Root (random-root-checked 5))

(mm-print (bfs Root 20))
(mm-print (ids Root 20))
(mm-print (dfs Root 5))
(mm-print (dfs Root 20))
```

Using `mm-print`*, searches are printed as tree-plots for Mathematica. In these tree-plots,* `x->y` *indicates an edge between two arbitrarily numbered vertices x and y.*

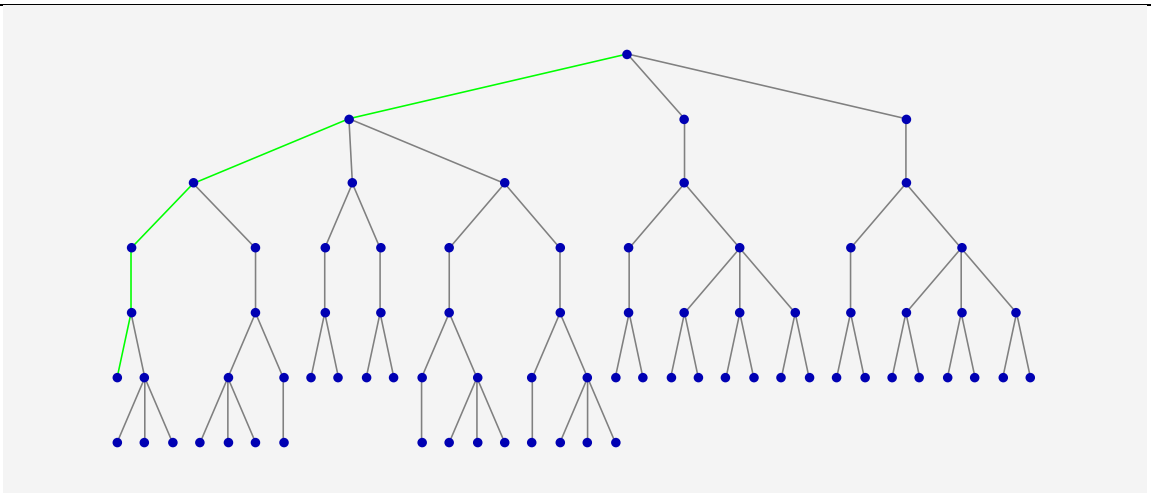
So although the optimal solution is 5 moves deep, any path under 21 moves will also be accepted. To only accept the optimal solution `GOAL-DEPTH` would be set to 5.

To accommodate suboptimal paths notice in most of the searches the depth-limit has been extended from 5 to 20. From here on, Search-*x* will denote a search with a depth limit of *x* beyond the optimal solution. So `(bfs Root 20)` will be denoted by BFS-15, because it will look $20-5=15$ nodes deeper than the optimal solution.

Here are the tree-plots generated from the above code:

BFS-15: The solution (green path) has been found. As it is only 5 edges long, it must be the optimal solution. Notice that nodes at depth 6 have been added to the tree because BFS has a depth-limit of 20. However, since BFS expands nodes in a first-in-first-out order, it discovers the solution before going any deeper.

```
Nodes expanded: 40
Tree size (nodes): 76
Closed-list size (states): 0
Search time (ms): 1
```



DFS: The DFS with a depth-limit at 5 not only finds the optimal solution (shown below), but does so without storing all the nodes in memory. When the depth-limit is set to the depth of the optimal solution, DFS outperforms all other naïve searches. However, when the depth-limit is too deep, problems will occur (see next search.)

0 Root

1	2	3
5	7	6
4		8

1 U

1	2	3
5		6
4	7	8

2 L

1	2	3
	5	6
4	7	8

3 D

1	2	3
4	5	6
	7	8

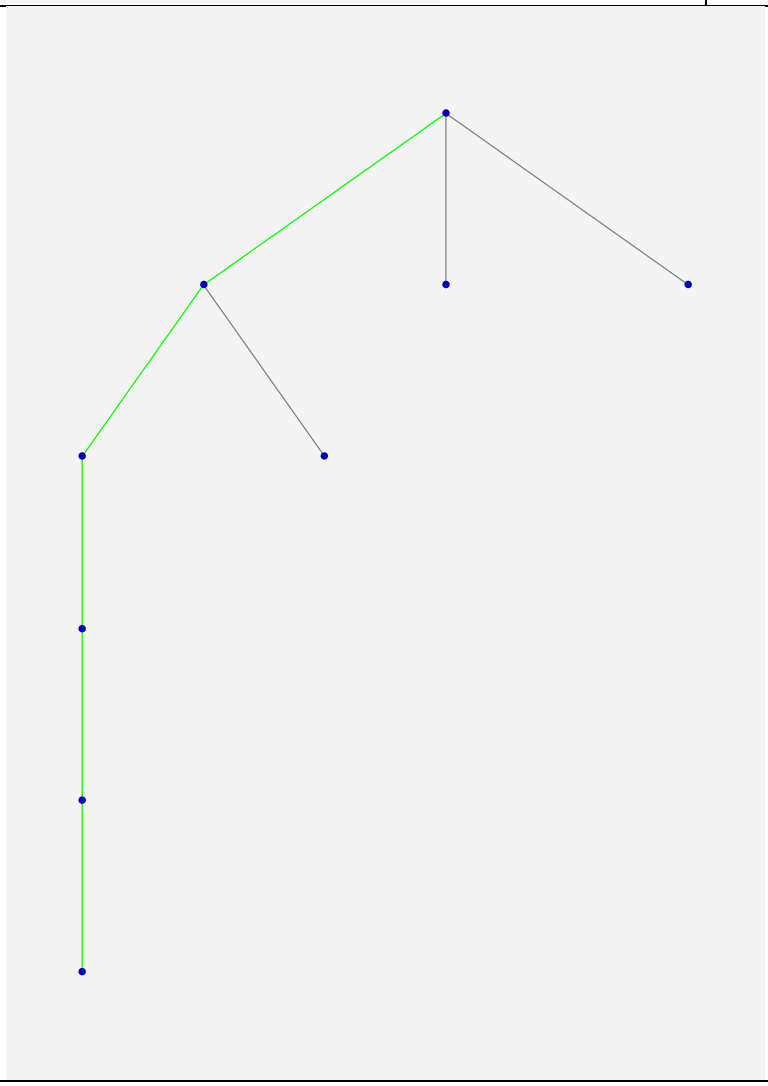
4 R

1	2	3
4	5	6
7		8

5 R

1	2	3
4	5	6
7	8	

```
Nodes expanded: 19
Tree size (nodes): 9
Closed-list size (states): 0
Search time (ms): 0
```



DFS-15: The solution found was much longer than the optimal one. This is to be expected because the `GOAL-DEPTH` has been relaxed to 20.

In addition, the performance is much worse—it is the only search that takes more than a millisecond to complete. And with `FILTER-CLOSED` set to false, the solution contains loops.

```
Nodes expanded: 10166
Tree size (nodes): 33
Closed-list size (states): 0
Search time (ms): 156
```

0 Root

1	2	3
5	7	6
4		8

1 U

1	2	3
5		6
4	7	8

2 U

1		3
5	2	6
4	7	8

3 L

	1	3
5	2	6
4	7	8

4 D

5	1	3
	2	6
4	7	8

5 R

5	1	3
2		6
4	7	8

6 U

5		3
2	1	6
4	7	8

7 L

	5	3
2	1	6
4	7	8

8 D

2	5	3
	1	6
4	7	8

9 R

2	5	3
1		6
4	7	8

10 U

2		3
1	5	6
4	7	8

11 L

	2	3
1	5	6
4	7	8

12 D

1	2	3
	5	6
4	7	8

13 D

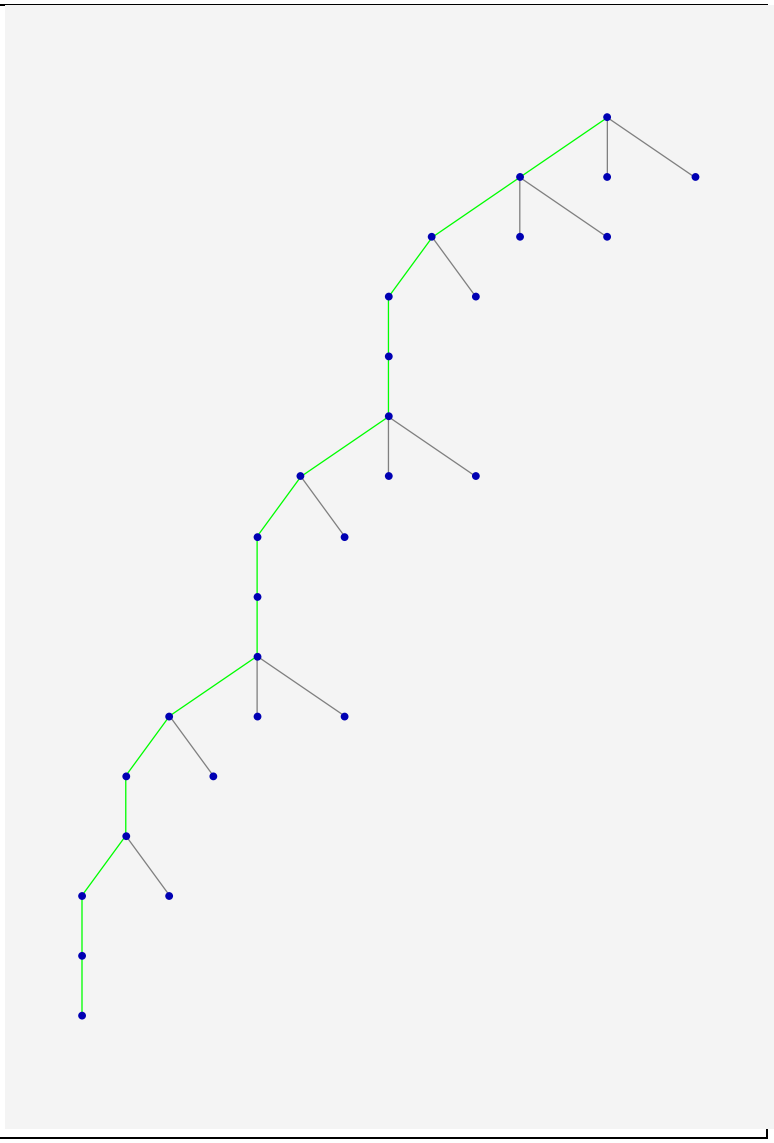
1	2	3
4	5	6
	7	8

14 R

1	2	3
4	5	6
7		8

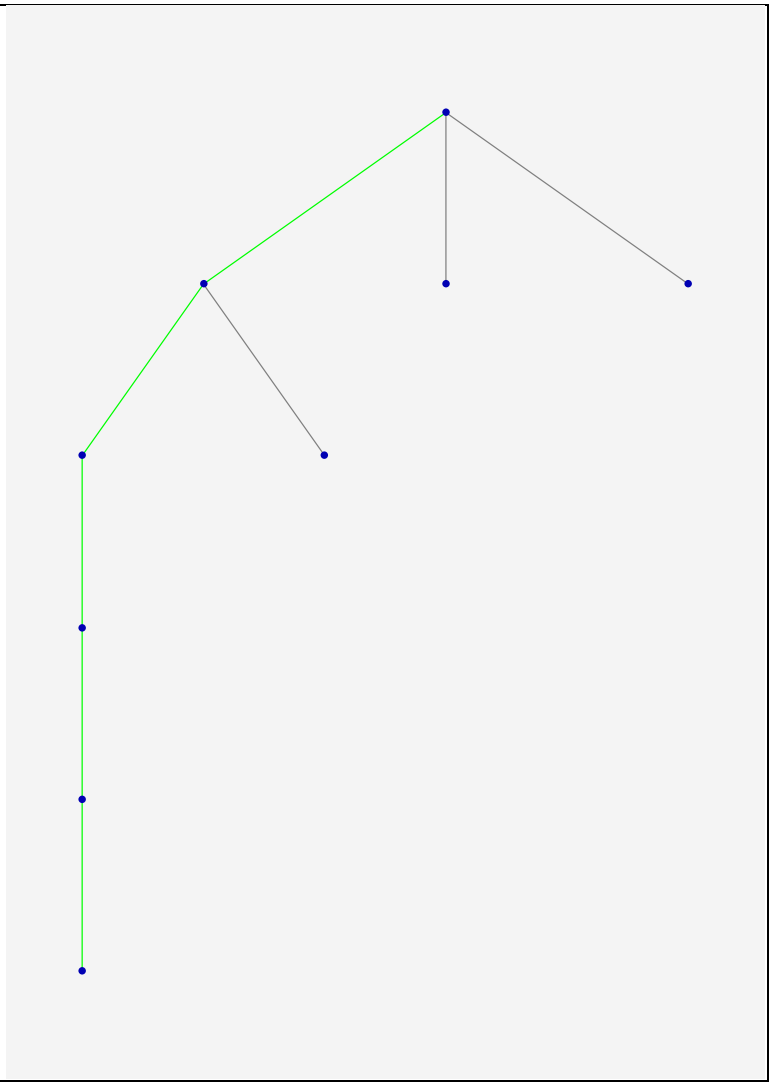
15 R

1	2	3
4	5	6
7	8	



IDS-15: The optimal solution is found without the need to store all the nodes in memory (the solution is the same as the BFS one). With each iteration, IDS runs a DFS with a depth-limit that increases. This is perhaps the best of both worlds. However if we know beforehand that the goal length is 5, we could just run DFS with a depth-limit of 5 and it would outperform IDS.

```
Nodes expanded: 86
Tree size (nodes): 9
Closed-list size (states): 0
Search time (ms): 0
```



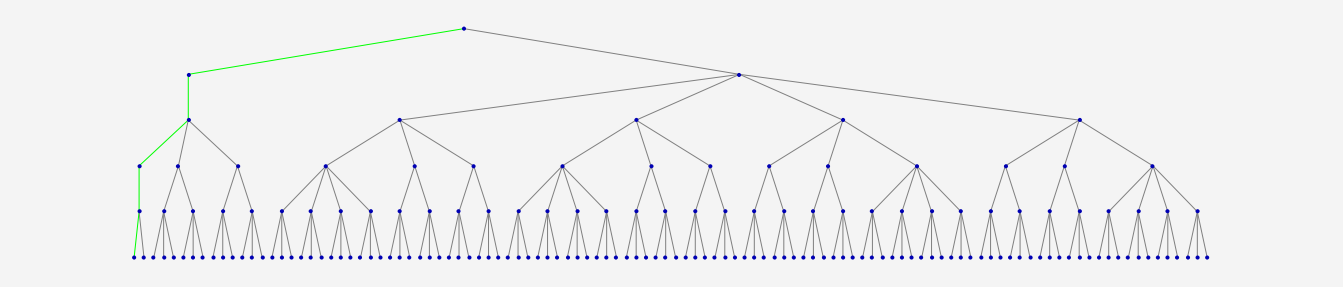
4.2 Closed-List Filtering

Up until this point, the search-space has been discussed as a tree-like structure, whereupon actions always lead to novel states. However, this is not the case with the n^2 puzzle, where it is common to encounter a "loop" that returns to a previous state. Up, right, down, left is one example.

A closed-list is naturally represented by a hash-table that uses states as keys and depths as values. States are added to the closed-list if they don't match a previous state (key). If they are found on the closed-list but are at a *lower* depth (value) they replace the old value. All nodes that don't modify the closed-list are not expanded.

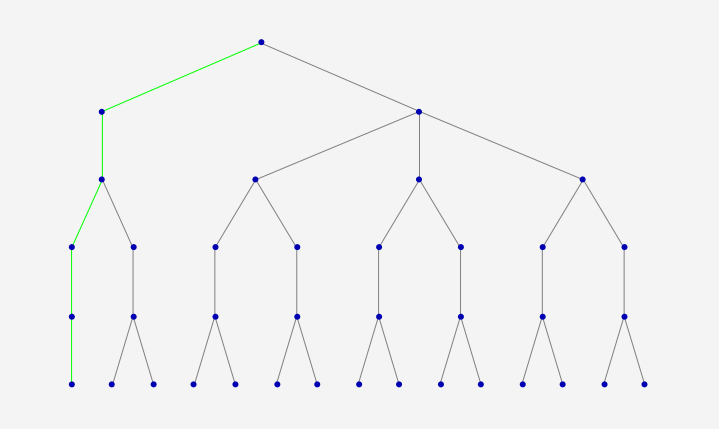
To demonstrate, a search tree *without* filtering:

```
(define FILTER-BACKWARDS #f)
(define FILTER-CLOSED #f)
(define Root (random-root-checked 5))
(mm-print (bfs Root 5))
```



is considerably denser than the exact same search with filtering

```
(define FILTER-CLOSED #t)
(mm-print (bfs Root 5))
```



The 170 nodes have been reduced to 38 without any other modification. Note that the hash table in this case has 61 entries, so the size reduction is not as dramatic as it seems.

5.0 Quantitative Results

With all the qualitative analysis out of the way, we turn back to the statistics mentioned in section 3.0 and see how all the modifications (depth-limit, backwards-filtering, closed-lists) affect the searches. Although most results turn out as expected, there are a couple surprising phenomena I'll describe in more detail.

5.1 Creating Data

To compare performance between searches, the function

```
(bench trials i max-depth search max)
```

will run a `search` on problems from depths `i` to `max-depth`, returned the results averaged over a number of `trials` and their standard deviation. If the desired depth-limit of the search is greater than the problem depth, then `max` will be that added depth. For instance,

```
(bench 100 1 3 dfs 5)
```

will run a DFS on problems of depths 1, 2, 3 with the depth-limit set to 6, 7, 8, respectively. The results are averaged over 100 trials. Printing the results in a readable form,

```
> (print (bench 100 1 3 dfs 5))

Finish 1; Finish 2; Finish 3;
Expanded = {48, 99, 176};
Tree = {3, 4, 5};
Closed = {49, 99, 175};
Time = {1, 2, 3};
ExpandedDev = {25.0, 69.0, 61.0};
TreeDev = {0.0, 1.0, 1.0};
ClosedDev = {25.0, 67.0, 59.0};
TimeDev = {1.0, 1.0, 1.0};
```

The output will be number of expanded nodes, tree size, closed-list size, and time (ms), as the problem size increases. `ExpandedDev` refers to the standard deviation of expanded nodes between trials. Results will be rounded.

5.2 Closed-Lists and Backwards Filtering

I'll now take a look at the advantages of closed-lists with or without backwards-filtering. I've used both a regular list and a hash table as closed-lists to compare the performance between them. Results were averaged over 100 trials. Searches in which in average time exceeded 200ms were cut-off. Space includes the closed-list and tree.

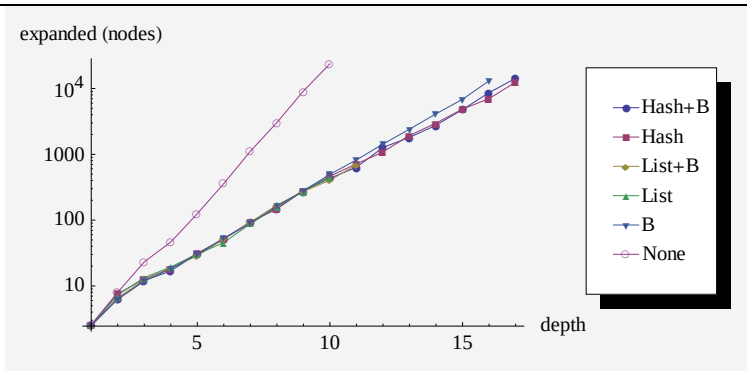
Explanation of graph legend:

- **Hash+B:** hash-table closed-list with backwards-filtering
- **Hash:** hash-table closed-list
- **List+B:** list-based closed-list with backwards-filtering
- **List:** list-based closed-list
- **B:** only backwards-filtering
- **None:** no filtering or closed-list used

DFS: Expanded Nodes

Not surprisingly, the number of expanded nodes is identical in all cases with a closed-list. Any state that would be backwards-filtered would also be added to a closed list. Backwards-filtering alone performs on-par with the closed-lists, although begins to lag behind (by an order of magnitude) as the chance of encountering a previously entered state increases with depth.

Note that some of the graphs are in log-linear scale.

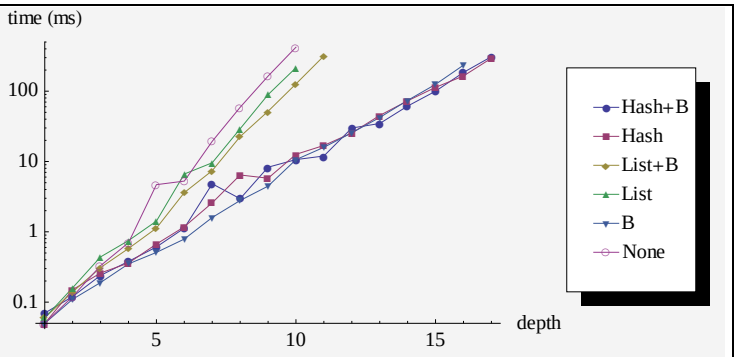


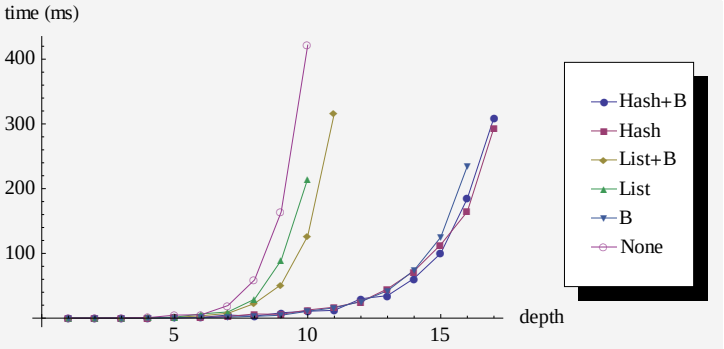
DFS: Time

The hash-table significantly outperforms the list. Because they are expanding and storing the same number of nodes, this can only be attributed to the constant vs. linear retrieval time of the two data structures.

Again backwards-filtering alone performs on-par with the closed-lists until depth increases.

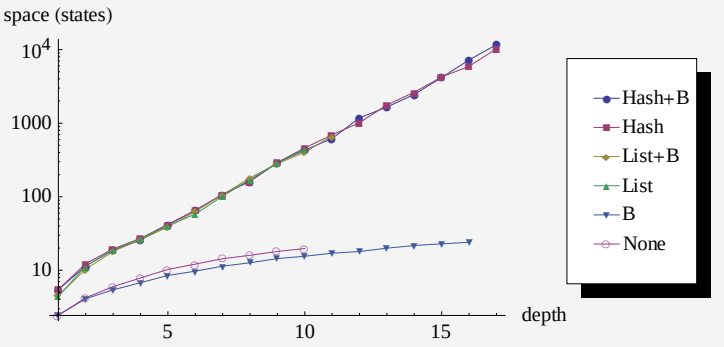
Although it appears that hash-table alone is better than hash-table with backwards-filtering, I've verified experimentally that the difference is statistically insignificant.





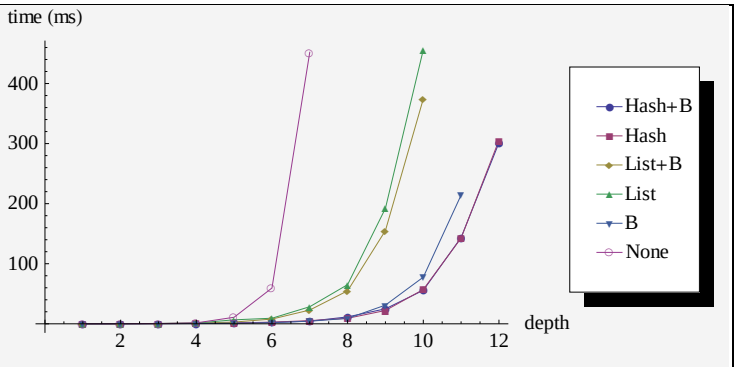
DFS: Space

Adding a closed-list gives DFS exponential space requirements. The searches without a closed-list are polynomial in space requirements with a slight advantage given by backwards-filtering. Note the use of log-linear scale.

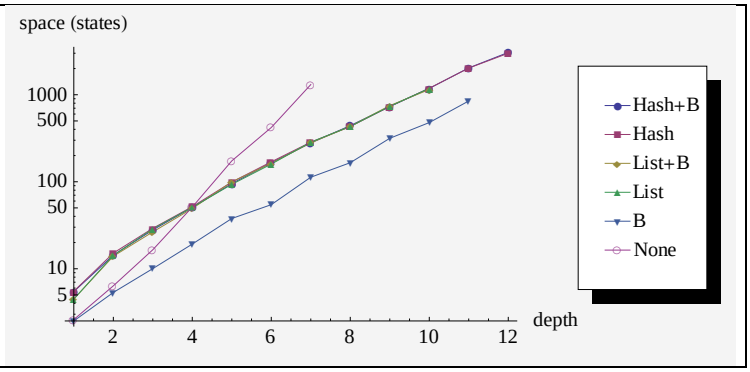


BFS: Expanded Nodes/Time

The pattern is similar to the DFS case.



BFS: Space
The BFS differs from the DFS in that all searches take exponential space. Filtering reduces the branching factor and thus takes less space. Backwards filtering alone takes the least amount of space because the closed-list contains no nodes. However, as depth increases the closed-list becomes more important in reducing tree size.



Taken together, the results suggest that backwards-filtering alone will reduce space-complexity the most (although this may not be true at greater depths with BFS), and a hash-table alone reduces time-complexity the most. Implementing the closed-list as a list instead of a hash-table is actually worse than having no closed-list at all (backwards-filtering assumed.)

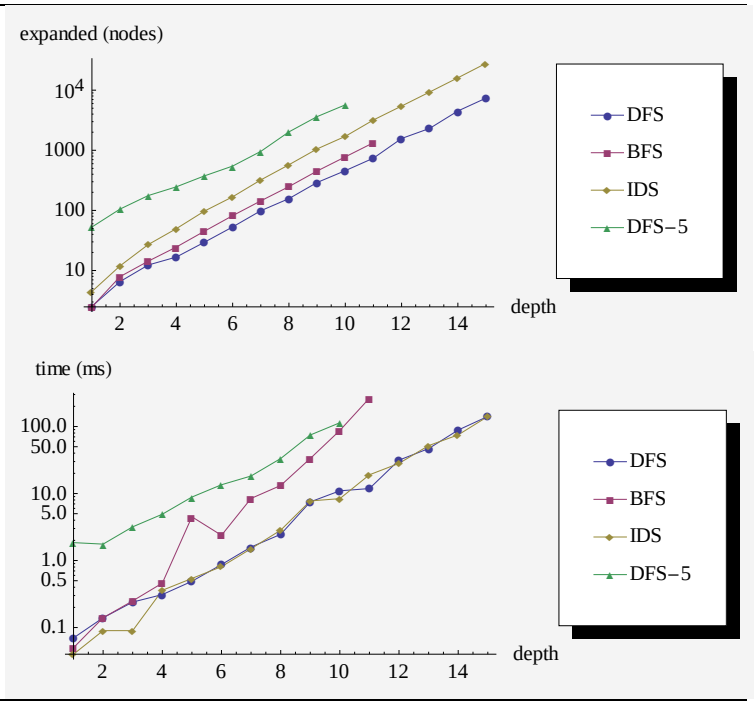
5.3 Head-to-Head Search Comparisons

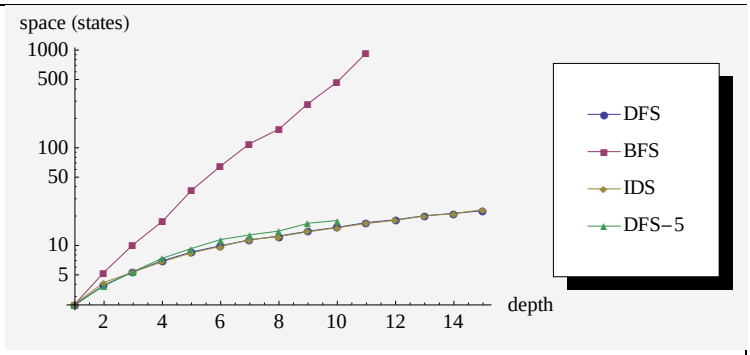
Now to compare searches with each other:

DFS-5 expands the most nodes because it is going 5 nodes deeper than any other search. Not surprisingly, IDS follows it because it is the summation of DFS at different levels. However, IDS manages to find solutions in the same time as DFS.

With respect to time, BFS performs worse than any other search by an order of magnitude, overtaking the DFS-5 at depth 10.

BFS has exponential space requirements, while the other searches have only polynomial requirements.





5.4 Backwards Filtering Worsens DFS-15?

Intuitively, filtering out backwards moves should always improve search times. However, I found at least one instance where this is not the case—when performing a depth-first search with a depth limit greater than the shallowest solution. Results were obtained by running DFS-10 on problems with a solution at depth 5 and averaging over 1000 trials.

Search	Time (ms)	Nodes Expanded	Std. Deviation (Nodes)	Tree
DFS-10	4	185	147	40
DFS-10 (backwards filtered)	30	1818	1157	21
DFS-10 (filtered + random successor)	47	2454	2242	22
DFS-10 (random successor)	88	4779	9870	36

Leaving in backwards moves forces DFS to spend more time in the local state space. Filtering out backwards moves plunges the search towards states which are highly unlikely to match the goal.

Interestingly, randomizing successor selection seems to negate this benefit completely and performs far worse than both the filtered and unfiltered DFS. One hint at why this might be the case comes from the ratio of the standard deviation of nodes expanded to the average nodes expanded. In the case of the random successor the ratio is 9870:4770 or nearly 2:1—much larger than in the other cases. The random order in which a list of successors is generated determines whether the search will stay at a shallow state (by making a backwards move) or will move even farther away from the goal. Randomized successors don't affect a *filtered* list because nearly every move is farther away from the goal.

As final proof, the tree size of the unfiltered searches is about twice as long because most solutions involve a lot of backtracking. Thus, filtering out moves is undesirable if a solution exists at a shallower level than the depth-limit.

5.5 Deepest Problems

The deepest problems are the reverse of the goal state. To prove this take,

0 Root

	8	7
6	5	4
3	2	1

0 Root

8		7
6	5	4
3	2	1

0 Root

6	8	7
	5	4
3	2	1

If the leftmost state (reverse of goal state) is indeed the deepest problem, then all of its successors must be *closer* to the solution. Running IDS on all three results in solutions of depths 28, 27, 27, respectively.

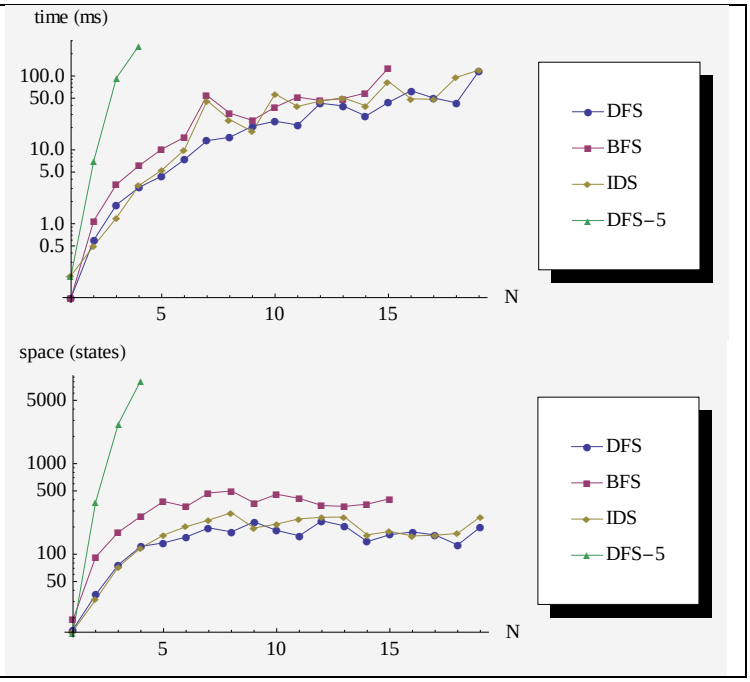
5.6 Varying *N*

I mentioned earlier in section 3.1 that changing *N* independently of *depth* does not affect search difficultly. To demonstrate:

All searches were conducted with solutions at depth 5. Up until $N = 5$ complexity increases exponentially with respect to *N*. However, beyond this time increases linearly and space stays constant with respect to *N*.

Manipulating *N* only affects search difficulty to the extent that it increases the branching factor on the way to finding the minimal solution.

Of course, with a high depth-limit as in DFS-5, the branching factor can cripple the search.



6.0 Raw Output

Here I include some briefly annotated output generated in previous sections, but that was either omitted from those sections or replaced by corresponding graphics.

4.1 / 4.2

The search trees are output as tree-plots for visualization in *Mathematica*. `1->2` indicates an edge drawn between two arbitrarily numbered vertices.

Input

```
(define FILTER-BACKWARDS #t)
(define FILTER-CLOSED #f)
(define Root (random-root-checked 5))

(mm-print (bfs Root 20))
(mm-print (ids Root 20))
(mm-print (dfs Root 5))
(mm-print (dfs Root 20))
```

Output for `bench-s`

```
TreePlot[{1->2,2->4,4->6,6->8,8->0,1->2,2->4,4->6,6->8,8->0,21->22,22->4,4->6,6->8,8->0,31->22,22->4,4->6,6->8,8->0,41->42,42->44,44->46,46->48,48->50,50->0,53->54,54->44,44->46,46->48,48->50,50->0,65->54,54->44,44->46,46->48,48->50,50->0,77->54,54->44,44->46,46->48,48->50,50->0,89->90,90->92,92->94,94->48,48->50,50->0,101->102,102->92,92->94,94->48,48->50,50->0,113->102,102->92,92->94,94->48,48->50,50->0,125->102,102->92,92->94,94->48,48->50,50->0,137->138,138->140,140->142,142->144,144->50,50->0,149->138,138->140,140->142,142->144,144->50,50->0,161->138,138->140,140->142,142->144,144->50,50->0,173->174,174->140,140->142,142->144,144->50,50->0,185->186,186->188,188->190,190->144,144->50,50->0,197->186,186->188,188->190,190->144,144->50,50->0,209->186,186->188,188->190,190->144,144->50,50->0,221->222,222->188,188->190,190->144,144->50,50->0,233->234,234->236,236->238,238->240,240->50,50->0,245->234,234->236,236->238,238->240,240->50,50->0,257->234,234->236,236->238,238->240,240->50,50->0,269->270,270->236,236->238,238->240,240->50,50->0,281->282,282->284,284->286,286->240,240->50,50->0,293->282,282->284,284->286,286->240,240->50,50->0,305->282,282->284,284->286,286->240,240->50,50->0,317->318,318->284,284->286,286->240,240->50,50->0,329->330,330->332,332->334,334->336,336->338,338->0,341->330,330->332,332->334,334->336,336->338,338->0,353->330,330->332,332->334,334->336,336->338,338->0,365->366,366->332,332->334,334->336,336->338,338->0,377->378,378->380,380->382,382->336,336->338,338->0,389->390,390->380,380->382,382->336,336->338,338->0,401->402,402->404,404->382,382->336,336->338,338->0,413->414,414->404,404->382,382->336,336->338,338->0,425->426,426->428,428->382,382->336,336->338,338->0,437->438,438->428,428->382,382->336,336->338,338->0,449->450,450->452,452->454,454->6,6->8,8->0,461->450,450->452,452->454,454->6,6->8,8->0,473->450,450->452,452->454,454->6,6->8,8->0,485->486,486->452,452->454,454->6,6->8,8->0,497->498,498->500,500->4,4->6,6->8,8->0,509->510,510->500,500->4,4->6,6->8,8->0,521->522,522->2,2->4,4->6,6->8,8->0,1->2},Automatic,0,EdgeRenderingFunction->(If[First[#2]<11 && Last[#2]<11,{Green, Line[#1]},{Gray, Line[#1]}] &), MultiedgeStyle->False]
TreePlot[{1->2,2->4,4->6,6->8,8->0,1->2,2->4,4->6,6->8,8->0,21->4,4->6,6->8,8->0,1->2},Automatic,0,EdgeRenderingFunction->(If[First[#2]<11 && Last[#2]<11,{Green, Line[#1]},{Gray, Line[#1]}] &), MultiedgeStyle->False]
TreePlot[{1->2,2->4,4->6,6->8,8->0,1->2,2->4,4->6,6->8,8->0,21->4,4->6,6->8,8->0,1->2},Automatic,0,EdgeRenderingFunction->(If[First[#2]<11 && Last[#2]<11,{Green, Line[#1]},{Gray, Line[#1]}] &), MultiedgeStyle->False]
TreePlot[{1->2,2->4,4->6,6->8,8->10,10->12,12->14,14->16,16->18,18->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,1->2,2->4,4->6,6->8,8->10,10->12,12->14,14->16,16->18,18->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,77->4,4->6,6->8,8->10,10->12,12->14,14->16,16->18,18->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,113->12,12->14,14->16,16->18,18->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,141->18,18->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,163->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,183->20,20->22,22->24,24->26,26->28,28->30,30->32,32->34,34->36,36->0,203->26,26->28,28->30,30->32,32->34,34->36,36->0,217->28,28->30,30->32,32->34,34->36,36->0,229->28,28->30,30->32,32->34,34->36,36->0,241->34,34->36,36->0,247->36,36->0,251->36,36->0,255->0,257->0,1->2},Automatic,0,EdgeRenderingFunction->(If[First[#2]<39 && Last[#2]<39,{Green, Line[#1]},{Gray, Line[#1]}] &), MultiedgeStyle->False]
```

5.2 / 5.3

`bench-4` and `bench-s` call `bench` with various flags to create the graphical output for *Mathematica*.

Input

```
(bench-4 100 "DFS" 1 999 dfs 0)
(bench-4 100 "BFS" 1 999 bfs 0)

(bench-s 100 1 999)
```

Output for `bench-s`

```
(*1*) (*2*) (*3*) (*4*) (*5*) (*6*) (*7*) (*8*) (*9*) (*10*) (*11*) (*12*) (*13*) (*14*) (*15*) (*16*)
DFSExpanded = {8/3, 11, 15, 65/3, 43, 203/3, 173/3, 169/3, 1024/3, 1051/3, 1147/3, 3539/3, 1183, 3103, 4880, 18293/3};
DFSTree = {7/3, 3, 16/3, 7, 23/3, 9, 40/3, 14, 38/3, 15, 52/3, 59/3, 64/3, 65/3, 23, 74/3};
DFSClosed = {3, 11, 49/3, 71/3, 134/3, 209/3, 63, 184/3, 335, 1007/3, 1133/3, 1097, 3311/3, 8146/3, 4267, 15752/3};
DFSTime = {0, 0, 1/3, 1/3, 2/3, 5/3, 1, 1, 7, 22/3, 25/3, 27, 74/3, 212/3, 283/3, 361/3};
DFSExpandedDev = {0.4714045207910317, 2.8284271247461903, 2.943920288775949, 8.956685895029603, 11.860297916431833, 14.817407180595247, 16.213848676020241, 41.346772008895915, 41.282226468811274, 234.78974045350068, 270.0152259081369, 514.431941292745, 253.262709454037706, 1251.649311908092, 1516.9620957690406, 3182.9033217418896};
DFSTreeDev = {0.4714045207910317, 0, 0.4714045207910317, 1.4142135623730951, 0.4714045207910317, 0.8164965809277260, 1.247219128924647, 0.8164965809277260, 1.247219128924647, 1.632993161855452, 2.0548046676563256, 0.9428090415820634, 0.4714045207910317, 2.0548046676563256, 1.632993161855452, 1.247219128924647};
DFSClosedDev = {0, 2.8284271247461903, 2.494438257849294, 7.542472332656507, 11.440668201153676, 14.007934259633796, 15.121728296285006, 41.10420362390634, 37.47888294315436, 217.35429346780543, 257.1346383166263, 451.7056563737054, 226.527898698788916, 1087.806763885735, 1278.704292112397, 2578.9353531167777};
DFSTimeDev = {0, 0, 0.4714045207910317, 0.4714045207910317, 0.4714045207910317, 0.4714045207910317, 0.8164965809277260, 0.8164965809277260, 0.8164965809277260, 4.109609335312651, 5.436502143433364, 12.569805089976535, 4.189935029992178, 27.157974069424906, 30.003703475108224, 62.993826858051911};
DFSTotalSpace = DFSTree + DFSClosed;

(*1*) (*2*) (*3*) (*4*) (*5*) (*6*) (*7*) (*8*) (*9*) (*10*) (*11*) (*12*)
BFSExpanded = {8/3, 5, 16, 64/3, 118/3, 262/3, 482/3, 195, 398, 2050/3, 3173/3, 1631};
BFSTree = {7/3, 17/3, 29/3, 22, 155/3, 209/3, 173/3, 692/3, 715/3, 61, 1759/3, 2944/3};
BFSClosed = {3, 7, 19, 31, 61, 116, 183, 291, 493, 706, 1295, 6089/3};
BFSTime = {0, 0, 1/3, 2/3, 4/3, 3, 14/3, 43/3, 24, 52/3, 281/3, 1052/3};
BFSExpandedDev = {0.4714045207910317, 1.4142135623730951, 1.4142135623730951, 6.128258770283412, 5.436502143433364, 18.803073034893938, 10.842303978193728, 46.00724580614087, 72.8880420005001, 15.173075568988056, 153.72341685276479, 372.0510717630041};
BFSTreeDev = {0.4714045207910317, 1.8856180831641267, 3.2998316455372216, 11.313708498984761, 9.741092797468305, 12.657891697365017, 24.63511495586917, 22.425184255405547, 175.46382976430087, 34.18576701884377, 373.0963533578007, 334.803756782321};
BFSClosedDev = {0, 0, 0, 0, 0, 18.384776310850235, 0, 32.526911934581186, 0, 0, 0, 254.08703670636606};
BFSTimeDev = {0, 0, 0.4714045207910317, 0.4714045207910317, 0.4714045207910317, 0.4714045207910317, 0.8164965809277260, 0.4714045207910317, 1.247219128924647, 9.092121131323903, 0.4714045207910317, 47.59084879353266, 151.28853962617995};
BFSTotalSpace = BFSTree + BFSClosed;

(*1*) (*2*) (*3*) (*4*) (*5*) (*6*) (*7*) (*8*) (*9*) (*10*) (*11*) (*12*) (*13*) (*14*) (*15*)
IDSExpanded = {14/3, 16, 82/3, 167/3, 100, 162, 1243/3, 1715/3, 2843/3, 5197/3, 8573/3, 13124/3, 24190/3, 31970/3, 65488/3};
IDSTree = {7/3, 13/3, 5, 7, 25/3, 26/3, 9, 41/3, 15, 44/3, 49/3, 58/3, 18, 62/3, 64/3};
IDSClosed = {3, 31/3, 37/3, 24, 97/3, 152/3, 532/3, 131, 671/3, 512, 2204/3, 3673/3, 6239/3, 6028/3, 15575/3};
IDSTime = {0, 0, 2/3, 0, 8/3, 17/3, 16/3, 0, 53/3, 79/3, 140/3, 134/3, 631/3};
IDSExpandedDev = {0.4714045207910317, 2.8284271247461903, 3.2998316455372216, 2.86744175568086756, 7.07106781186547755, 29.87752778706208, 6.79869268479038, 88.02019970186136, 131.49735950031683, 507.1234782794248, 343.3504850084758, 185.069956743089140, 201.06604774440555, 1668.1195667244267, 4228.7884264355855};
IDSTreeDev = {0.4714045207910317, 0.9428090415820634, 0, 1.4142135623730951, 0.4714045207910317, 0.9428090415820634, 0.8164965809277260, 0.4714045207910317, 0.8164965809277260, 1.6996731711975955, 1.2472191289246475, 1.6996731711975955, 1.4142135623730951, 1.2472191289246475, 2.8674417556808756};
IDSClosedDev = {0, 1.8856180831641267, 3.2998316455372216, 7.8740078740118111, 6.5996632910744443, 29.101355447622866, 6.018490028422596, 55.80919876388360, 125.33244680537527, 345.05168694945786, 312.54795187656436, 150.88700702475641, 165.249978988806387, 1406.903060705399, 3508.705775955319};
IDSTimeDev = {0, 0, 0, 0.4714045207910317, 0, 0.8164965809277260, 1.8856180831641267, 2.4944382578494294, 2.8674417556808756, 0, 6.342099196813483, 4.109609335312651, 1.6996731711975955, 31.668421004036322, 198.153700164515955};
IDSTotalSpace = IDSTree + IDSClosed;

(*1*) (*2*) (*3*) (*4*) (*5*) (*6*) (*7*) (*8*) (*9*) (*10*) (*11*)
DFS5Expanded = {62, 493/3, 500/3, 655/3, 941/3, 1954/3, 3347/3, 1885, 3661/3, 2900, 25832/3};
DFS5Tree = {7/3, 3, 5, 25/3, 25/3, 31/3, 14, 44/3, 61/3, 50/3, 56/3};
```

```
DFS5Closed = {187/3, 162, 499/3, 213, 302, 1828/3, 3080/3, 1700, 3355/3, 2559, 21499/3};
DFS5Time = {4/3, 13/3, 10/3, 5, 19/3, 14, 70/3, 113/3, 76/3, 70, 829/3};
DFS5ExpandedDev = {24.041630560342615, 24.98443960192468, 44.78342947514801, 151.9875725913873, 214.48905695991942, 376.3529667154964, 674.5429728506718, 928.3996984058106, 764.9890340535753,
1897.1432910211781, 5314.632965271972};
DFS5TreeDev = {0.4714045207910317, 0, 0, 2.624669291337270 2, 0.9428090415820634, 1.699673171197595, 3.559026084010437, 3.7712361663282534, 3.2998316455372216, 3.681787005729087,
2.8674417556808756};
DFS5ClosedDev = {23.570226039551585, 24.041630560342615, 42.897811391983886, 143.78687932724137, 196.5756851698602, 341.91259441884006, 602.5503206279861, 816.4022701258655, 680.0403582794839,
1639.4446214089292, 4288.65125133247};
DFS5TimeDev = {0.4714045207910317, 0.4714045207910317, 0.4714045207910317, 4.320493798938574, 4.0276819911981905, 8.286535263104035, 14.055445761538676, 17.913371790059205, 16.131404843417148,
47.588514020367, 227.08931184790612};
DFS5TotalSpace = DFS5Tree + DFS5Closed;

ListLogPlot[{DFSTime, BFSTime, IDSTime, DFS5Time}, PlotRange -> All, Joined -> True, PlotMarkers -> {Automatic, Tiny}, AxesLabel -> {"depth", "time (ms)"}, PlotLegend -> {"DFS", "BFS", "IDS", "DFS-
5"}, LegendPosition -> {1, -.4}, LegendBorderSpace -> 1]

ListLogPlot[{DFSTotalSpace, BFSTotalSpace, IDSTotalSpace, DFS5TotalSpace}, PlotRange -> All, Joined -> True, PlotMarkers -> {Automatic, Tiny}, AxesLabel -> {"depth", "space (states)"}, PlotLegend -
> {"DFS", "BFS", "IDS", "DFS-5"}, LegendPosition -> {1, -.4}, LegendBorderSpace -> 1]

ListLogPlot[{DFSExpanded, BFSExpanded, IDSExpanded, DFS5Expanded}, PlotRange -> All, Joined -> True, PlotMarkers -> {Automatic, Tiny}, AxesLabel -> {"depth", "expanded (nodes)"}, PlotLegend ->
{"DFS", "BFS", "IDS", "DFS-5"}, LegendPosition -> {1, -.4}, LegendBorderSpace -> 1]
```

5.4 Backwards Filtering Worsens DFS-15?

When passed to `print`, the output of `bench` is easy enough to interpret.

Input

```
(define FILTER-BACKWARDS #f)
(define FILTER-CLOSED #f)
(define SHUFFLE-SUCCESSORS #f)
(print (bench 1000 5 5 dfs 10))

(define FILTER-BACKWARDS #t)
(print (bench 1000 5 5 dfs 10))

(define SHUFFLE-SUCCESSORS #t)
(print (bench 1000 5 5 dfs 10))

(define FILTER-BACKWARDS #f)
(print (bench 1000 5 5 dfs 10))
```

Output

```
(*5*)(*6*)
Expanded = {208};
Tree = {41};
Closed = {0};
Time = {6};
ExpandedDev = {163.0};
TreeDev = {5.0};
ClosedDev = {0};
TimeDev = {26.0};
TotalSpace = Tree + Closed;

(*5*)(*6*)
Expanded = {1808};
Tree = {21};
Closed = {0};
Time = {34};
ExpandedDev = {1125.0};
TreeDev = {5.0};
ClosedDev = {0};
TimeDev = {48.0};
TotalSpace = Tree + Closed;

(*5*)(*6*)
Expanded = {2067};
Tree = {22};
Closed = {0};
Time = {44};
ExpandedDev = {2102.0};
TreeDev = {5.0};
ClosedDev = {0};
TimeDev = {60.0};
TotalSpace = Tree + Closed;

(*5*)(*6*)
Expanded = {4223};
Tree = {37};
Closed = {0};
Time = {91};
ExpandedDev = {7250.0};
TreeDev = {4.0};
ClosedDev = {0};
TimeDev = {163.0};
TotalSpace = Tree + Closed;
```